Escola de Engenharia
Universidade do Minho

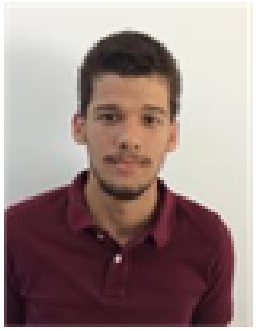
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
**Mestrado Integrado em Engenharia Informática**
***Sistemas Distribuídos***

# *SERVERUM*

## *Alocação de servidores na nuvem*

## Grupo 8

Adriana Meireles
A82582



Nuno Silva A78156

Ricardo Pereira A73577

Shahzod Yusupov
A82617

Braga, 29 de Dezembro de 2018

# Conteúdo

# 1. Introdução

Na unidade curricular de Sistemas Distribuídos, foi-nos proposto um problema bastante interessante que aborda a interação entre "servidores"e "clientes"de um determinado sistema. De facto, encontramo-nos num mundo digital onde essas ligações são indispensáveis. No entanto, por trás de todos estes processos existe uma infraestrutura que necessita de estabelecer algumas normas e padrões para que possam existir essas comunicações. A matéria lecionada nesta unidade curricular, trata esse tema e permite-nos compreeender como é que os nossos programas podem comunicar de uma forma eficiente com outras máquinas, também elas programadas! Conceitos chave como criação de threads, gestão de concorrência, mecanismos de exclusão mútua, entre outros, são essenciais para podermos trabalhar com máquinas "clientes"e "servidores".

De uma forma geral, foi-nos pedida a criação de um sistema em que existisse um servidor que fizesse a atribuição de servidores virtuais aos vários utilizadores que os solicitassem. A decisão de um servidor virtual ser ou não atribuido a um dado utilizador depende do primeiro estar ou não a ser utilizado. Não estando, pode ser reservado pelo seu preço nominal ou poderá ir a um leilão em que entram vários clientes! Estando em utilização, quem o solicita terá que aguardar que o outro utilizador o liberte ou então terá acesso imediato caso o servidor tenha sido licitado, desconectando o outro utilizador.

Tendo em conta as informações anteriores, delineamos um plano geral daquilo que teríamos que fazer e ter atenção. Decidimos de que forma seria mais prático e mais eficiente guardar a informação que era introduzida, como os dados dos utilizadores e a que devia existir no sistema por definição, por exemplo, dados dos servidores. Após termos bem definidas todas as estruturas, assim como as classes, começamos a fazer a aplicação dos *locks* que os objetos deveriam ter que são indispensáveis no controlo de concorrência.

# 2. Desenvolvimento do projeto

Após uma leitura cuidada e intensiva do enunciado, debatemos uns com os outros aquilo que entendemos do enunciado. De uma forma geral,teríamos um servidor que aceitava a conexão de vários clientes que após o registo, se necessário, iniciavam sessão, ficando aptos para poderem comprar a reserva de um servidor virtual ou então para licitar o seu preço.

Posto isto, conseguimos fasear o projeto de modo a melhorar a compreensão do problema. Ora, inevitavelmente, a fase inicial teria de ser a criação das estruturas nas quais armazenamos informação a que teríamos de ter acesso tão rápido e prático quanto possível. Esta fase engloba também a construção dos menus com que o cliente vai interagir, implementação do código que permitirá os inícios de sessão e as comunicações entre clientes e o servidor.

Quanto à segunda fase, achamos que deveria ser aquela em que o utilizador escolhe um servidor pelo preço nominal, ficando com ele enquanto o desejar. Para além das exceções inerentes a esta tarefa era requerido ainda que um cliente pudesse conectar-se a um servidor em uso que tenha sido licitado desde que o cliente em questão se predispusesse a pagar o preço nominal do servidor (algo que trará mais lucro à empresa).

A terceira fase, a mais complicada, a nosso ver, foi também a que melhor nos ajudou a perceber a necessidade de todos estes conceitos, visto que, mais que em qualquer outra, é necessário gerir as consultas e as alterações feitas das variáveis que podem estar a ser acedidas por um número muito elevado de clientes.

Obviamente, estas tarefas teriam que ser construidas dentro da classe que corresponde ao código do servidor. Esta entidade é aquela que gere todo o processo e como tal, é a que decide o que é que cada cliente pode ou não fazer. Assim, a classe de que falamos é a *AuctionServerTHD*, da qual se criarão tantas instâncias quantos os clientes que se conectarem ao servidor. Para fazer esta instanciação de servidores, que acabam por ser todos o mesmo, temos a classe *AuctionServer*, que é por sua vez instanciada pela função *main*;

A par desta última, existe outra entidade fundamental que é o cliente. A função do cliente, não é, nem mais, nem menos, do que escolher, consultar e utilizar aquilo que o servidor tem para lhe oferecer. Posto isto, para aproximar este trabalho prático da realidade, vamos ter outra função *main*, isto é, não vamos simplesmente criar clientes na mesma *main* em que criamos o servidor. Assim, instanciamos a class *BidderClient* que, de uma forma geral, lê aquilo que o utilizador escreve no *standard input* e transmite ao servidor. Nesta, é ainda criada uma instância de *BidderClientListener* que será uma *thread* que ficará encarregada de receber e escrever no *standard output* tudo o que o servidor lhe envia! Passemos então a uma explicação mais detalhada daquilo que foi feito em cada uma destas fases.

## 2.1 Primeira fase - Classes, base de dados e comunicação

### 2.1.1 Classes e Base de Dados

Criamos a classe *Bidder*, que representa um cliente, com um nome de utilizador, uma *password* e um saldo. Tem ainda um *ReentrantLock* para o controlo do acesso às variáveis das intâncias que forem criadas.

A classe *Server*, representa um servidor virtual, com o seu nome, preço nominal, nome do utilizador a que está connectado (se estiver), o preço base de licitação, o preço mais alto licitado durante o leilão,

o tempo disponícel para receção de licitações, um *hashmap* com todos os licitantes do leilão que estiver a decorrer, cujas chaves são os nomes e os valors os *sockets* dos licitantes. Tal como a *Bidder*, esta classe tem um *ReentrantLock* e quatro variáveis de condição cuja existência será justificada durante a explicação das próximas fases.

As bases de dados estão presentes em qualquer sistema que envolva informação. Como este não foge à regra, criamos uma classe *DataBase* com três *hashmaps* chamados *bidders*, *connectedBidders* e *servers*. O primeiro guarda todos os clientes que se registaram na aplicação, o segundo, os clientes que estão ligados ao servidor e o terceiro, todos os servidores que existem. As chaves e os valores são, respetivamente, o nome do *bidder* e o *bidder*, o nome do bidder e a *socket* e nome do *server* e o *server*. Para esta classe, não criamos um, mas três *ReentrantLocks* para o controlo de acesso das várias *threads* do servidor a cada *hashmap*. Esta medida foi possível implementar depois de nos apercebermos que a grande maioria das operações feitas em cada *hashmap* são totalmente independentes, isto é, não há operações que envolvam a alteração/consulta do conteúdo de mais que um *hashmap*, operações que teriam que ser consideradas atómicas. Para esse caso, teríamos também a possibilidade de criar funções que fizessem os *locks* necessários e chamá-las quando necessárias. Ao fazer três locks em vez de um, **reduzimos extremamente a região crítica** criada para aceder à base de dados, isto é, uma operação que, a título de exemplo, queira apenas ir buscar o tempo de leilão de um servidor, nao precisa de bloquear toda a base de dados. Bloqueia apenas o acesso ao *hashmap* a que vai buscar essa informação. Além disso, fizemos ainda um encapsulamento que permite à *thread* criar regiões críticas durante o período de tempo em que são necessárias, como podemos ver nas imagens em baixo.

```
public double getServersServerNominalPrice(String servername){
    this.lockServers.lock();
    Server server = this.servers.get(servername);
    this.lockServers.unlock();
    return server.getNominalPrice();
}
```

**Figura 2.1:** função do tipo get ao nível da base de dados

```
public int getAuctionTime(){
    this.lock.lock();
    int result = this.auctionTime;
    this.lock.unlock();
    return result;
}
```

**Figura 2.2:** função do tipo get ao nível do servidor

### 2.1.2 Canais de comunicação

Esta, é uma tarefa *standard* sempre que temos programas que envolvam servidores e clientes. Estes canais de comunicação são criados com base no *socket* que se atribuiu ao servidor. Tecnicamente, o servidor fica à espera que um cliente se ligue ao seu porto, e nesse momento, cria-se um novo *socket* no servidor que fica associado a esse cliente. Do lado do cliente, acontece o mesmo processo. É criado um *socket* no cliente onde a *thread* do servidor irá ler e escrever a informação que dele vem. Esse canais abrem-se instanciando-se as classes *PrintWriter* para escrita e a *BufferedReader* para leitura.

Ainda nos canais de comunicação, lembre-se, é boa prática encerrar os descritores que foram abertos tal como os sockets, algo que nós fazemos na função *logout*.

### 2.1.3 Início de sessão

Muito sucintamente, o início de sessão não é mais que um menu onde se verifica a existência e/ou validade do nome de utilizador inserido pelo utilizador bem como da respetiva password.

## 2.2 Segunda fase - Reserva a pedido

Após ser feito o início de sessão, é imprimido um catálogo com todos os servidores, tantos os disponíveis como aqueles que estão em utilização. Depois, o utilizador pode optar por fazer uma reserva a pedido ou inicar/entrar um/num leilão. É a primeira que diz respeito a esta fase e despoleta-se com a invocação da função *demandInstance()*. Assim que começa a executar a função, a *thread* do servidor pede ao utilizador para introduzir o nome do servidor que deseja, executando uma das opções enunciadas em baixo, conforme o estado deste último. Há que relembrar que a verificação dos vários estados por parte da *thread* em questão, tem que ser feita de uma forma atómica, para que o estado do servidor escolhido não mude enquanto as várias verificações são feitas.

### 2.2.1 Servidor em uso após leilão

Caso a *thread* do servidor consiga entrar neste cenário, então quer dizer que o utilizador em questão quer reservar um servidor a pedido, no entanto, esse está a ser usado por outro utilizador que o reservou num leilão do qual saiu vencedor. Neste caso, as regras ditam que o atual utilizador deve ceder o acesso ao novo utilizador que o reservará pelo preço nominal. Este, é um dos casos cujo código exige um pouco mais de raciocínio para ser formulado. A solução que nós optamos por implementar passa pela declaração de uma variável de condição, *inConnectionSpot*. De um forma sucinta, permitimos que a *thread* associada àquele utilizador que procura o acesso, invoque a função que estabelece a conexão com o servidor, não obstante, essa função é declarada com um *if statement* que força a desconexão do servidor e atira a *thread* para um estado de espera. Entertanto, a outra *thread* inicia o processo de desconexão e assim que o concluir, envia um *signal* para a *thread* "adormecida"que fará a conexão normalmente como que se de uma reserva a pedido se tratasse.

### 2.2.2 Servidor em uso após reserva a pedido

Outro caso em que se usam variáveis de condição é quando existe um servidor virtual reservado a pedido. Caso o novo utilizador decida esperar pela desconexão do servidor, é atirado para um estado de espera, do qual sai quando a operação de desconexão for concluida.

```
public String establishConnectionDemand(String username){
    this.lock.lock();
    if(this.bidderName != null){
        try{
            this.inConnectionDemand.await();
        }catch(InterruptedException e){}
    }
    this.bidderName = username;
    String temporary = this.name;
    this.lock.unlock();
    return temporary;
}
```

**Figura 2.3:** função de conexão a um servidor

```
public void disestablishConnectionDemand(){
    this.lock.lock();
    this.bidderName = null;
    this.inConnectionDemand.signal();
    this.lock.unlock();
}
```

**Figura 2.4:** função de desconexão de um servidor

### 2.2.3 Servidor em leilão

Um servidor que esteja em leilão não pode ser reservado a pedido, e como tal, caso esse seja o cenário, a reserva é negada a esse utilizador.

### 2.2.4 Servidor livre

Por fim, se o utilizador selecionar um servidor que está livre, a conexão é feita sem qualquer entrave até que o utilizador decida abandoná-la!

## 2.3 Terceira fase - Reserva em leilão

De acordo com o que foi dito anteriormente, após o início de sessão ou desconexão de um servidor por parte de um utilizador, é-lhe pedido que escolha entre obtenção de um servidor a pedido ou a leilão (ou então consulta do valor em dívida). Se a opção escolhida for a reserva por leilão, mais uma vez, existem alguns cenários possíveis nos quais a *thread* pode "cair". Mas antes disso, é ainda verificada uma situação: se o utilizador que escolhe a opção do leilão para o servidor, é o primeiro a fazê-lo. Caso seja, inicia o leilão, caso contrário espera até que o leilão comece.

### 2.3.1 Servidor em uso após reserva a pedido

Um cliente, pode, por acaso, escolher um servidor que está a ser usado por outro cliente. Nesse caso, não há nada que possa fazer e a reserva é negada.

### 2.3.2 Servidor em uso após leilão

Tal como no caso anterior, o cliente terá que escolher outro servidor ou tentar até que consigo iniciar/entra um/num leilão. Um servidor virtual, após ser leiloado, apenas pode ser desconectado por uma reserva a preço nominal. Como não é o caso, a *thread* é forçada a abandonar a reserva.

### 2.3.3 Servidor livre

Estando o servidor livre, relembre-se, a *thread* ganha total controlo sobre o server, pois todas as outras que lhe tentam aceder, ficam à espera do início do leilão (devido a uma variável de condição). Neste processo, é ainda usada outra variável de condição bastante importante: *countDownInit*, que faz com que a *thread* que invoca o temporizador do leilão não avance até que o leilão inicie. Após o início do leilão, esta thread faz o mesmo que as *threads* que escolheram esta opção de reserva a leilão do servidor em questão.

### 2.3.4 Servidor em licitação

Após entrarem no leilão, as *threads* do servidor pedem aos respetivos clientes que digitem um número, correspondente a uma licitação. Esta operação é precedida por uma bateria de testes feitos ao número introduzido, nomeadamente se este número está dentro dos limites que é suposto e se está dentro do tempo estabelecido para o leilão. Nesta fase de verificação e alteração do valor de leilão, caso seja válido, é crucial a existência de exclusão mútua visto que enquanto existem *threads* a verificar o último valor mais alto licitado, outras estão a defini-lo. Desta forma, atomizamos também esta parte do leilão para que apenas uma *thread* a pudesse executar.

# 3. Conclusões

A realização de um projeto prático numa unidade curricular é sempre benéfica. Ajuda-nos a consolidar toda a matéria que aprendemos e, tão ou mais importante, dá-nos uma melhor perceção do impacto que o que estudamos tem em sistemas mais complexos.

Conseguimos perceber que a versatilidade dos programas acarreta bastantes problemas se não os escrevermos da forma mais correta. Exemplo disso, foi uma situação que nos aconteceu na realização deste trabalho prático: por lapso, permitimos que uma função permanecesse indefinidamente com a exclusão mútua de uma instância, causando um *deadlock*, *bug* que nos custou bastante tempo.

A auto-crítica negativa que achamos válida, prende-se com as declarações de algumas funções. Sabemos que temos casos em que as funções poderiam ter sido declaradas de forma menos confusa, mais eficiente e mais correta do o que foram. São más práticas como essas que levam muitas vezes à ocorrência de erros inesperados, bastante difíceis de solucionar, tal como nos aconteceu.

O nosso nível de satisfação é elevado, não só porque tivemos prazer em aprender mais acerca de sistemas distribuídos, mas porque foi um trabalho que nos "abriu as portas"para aquilo que nós realmente queremos conhecer e explorar cada vez mais, a interação constante com sistemas servidor/cliente.

## 3.1 Trabalho Futuro

Seria interessante realizar um outro projeto, que nos permitisse ainda mais a exploração destes recursos mas de forma ainda mais real, como por exemplo, um jogo online, preparado para vários jogadores. Perceber o que é necessário fazer para que se estabeleça conexões com outros utilizadores fora da nossa rede privada, é também uma meta a atingir.

**Nota:** Compilar e executar um cliente ou um servidor do SERVERUM

Para compilar tanto um cliente como um servidor, apenas será necessário, num terminal, entrar na respetiva pasta e introduzir o seguinte comando: **javac *.java**. Ainda no terminal, para executar um servidor é necessário introduzir o comando **java InitS <porto desejado>**. Para executarmos um cliente, obviamente, precisamos de um servidor em execução e de escrever no terminal o seguinte comando: **java InitC <ip do servidor> <porto desejado>**.